CAPÍTULO 14
HERANÇA

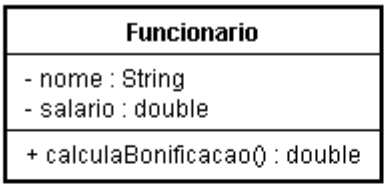
É um princípio da Programação Orientada a Objetos que permite que as classes compartilhem atributos e métodos comuns baseados em um relacionamento.

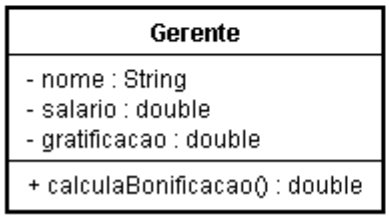
A herança possibilita a aplicação de vários conceitos de orientação a objetos que não seriam viáveis sem a organização e estruturação alcançada por sua utilização.

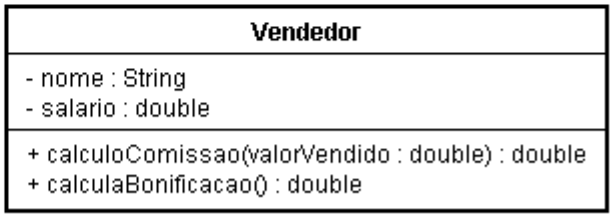
A habilidade de estender é considerada um dos maiores benefícios da orientação a objetos, ou seja, fazer uma subclasse, o comportamento de uma classe existente e continuar a usar o código escrito para a classe original ao atuar sobre uma instância da subclasse. A classe original é conhecida como superclasse. Quando você estende uma classe para criar uma nova classe, a nova classe estendida herda campos e métodos da superclasse.

A superclasse é também conhecida como "classe mãe" e a subclasse como "classe filha".

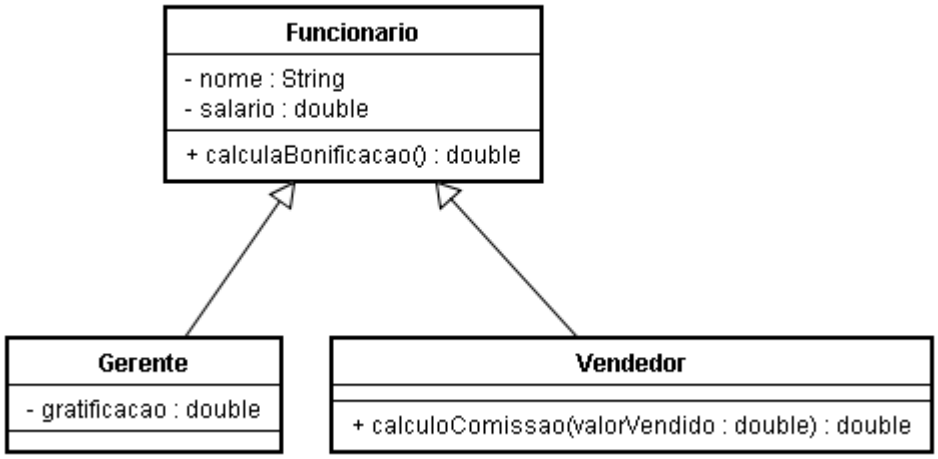
Observe as definições de funcionários, gerentes e vendedores de acordo com o diagrama de classes:







E agora observe com o uso da herança:



Os atributos e métodos comuns foram centralizados na classe Funcionario e compartilhados pelas classes Gerente e Vendedor.

Sem o recurso da herança, teríamos que replicar esses atributos e métodos nas três classes, procedimento desaconselhável em qualquer linguagem de programação, por trazer complexidades extras na manutenção e expansão dos sistemas. Por exemplo, vamos considerar que o método calcula bonificação foi atualizado com uma nova regra. Tal atualização deveria ser feita para o gerente e vendedor e, caso uma delas não fosse realizada, a atualização da bonificação ficaria inconsistente.

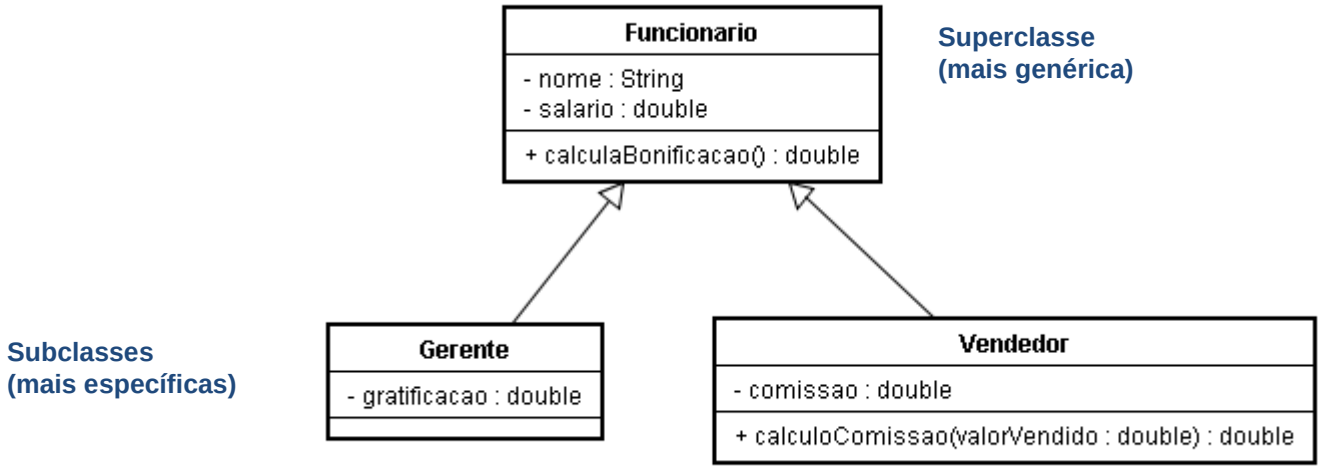
14.1. GENERALIZAÇÃO E ESPECIALIZAÇÃO

A **generalização** é o agrupamento de características (atributos) e regras (métodos) comuns em um modelo de sistema, então a superclasse contém o que é genérico da classe.

A classe Funcionario contém dois atributos e um método, os quais são comuns para gerentes e vendedores.

Já a **especialização** é o processo inverso, é a definição das particularidades de cada elemento de um modelo de sistemas, detalhando características e regras específicas de um o objeto, então a subclasse contém o que é específico da classe.

O gerente ganha gratificação e o vendedor ganha comissão das vendas, mas ambos têm nome, salario e ganham bonificação.



14.2. IMPLEMENTAÇÃO DA HERANÇA NO JAVA

Considere que todo fim de ano, os funcionários recebem uma bonificação. Todos os funcionários comuns recebem 30% do valor do salário. A herança é definida na implementação da subclasse, ou seja, quanto à codificação, a superclasse não apresenta nenhuma diferença.

```
public class Funcionario {

    private String nome;
    private double salario;

    // getters e setters omitidos

    public double calculaBonificacao() {
        return this.salario * 0.3;
    }
}
```

Em Java, é utilizada a cláusula ***extends*** na criação da classe (para mostrar que a classe Gerente herda atributos e métodos da classe Funcionario).

Podemos dizer que o objeto do tipo Gerente **estende** os atributos e métodos do objeto do tipo Funcionario, pois agora Gerente **é um** Funcionario.

```
public class Gerente extends Funcionario{

    private double gratificacao;

    // getters e setters omitidos

}
```

O mesmo acontece com a classe Vendedor. Podemos dizer que o objeto do tipo Vendedor **estende** os atributos e métodos do objeto do tipo Funcionario, pois agora Vendedor **é um** Funcionario.

```
public class Vendedor extends Funcionario {

    //método calcula comissão
}
```

14.3. VANTAGENS DA HERANÇA

- **Evitar duplicação de código** – Não reescrever o código.

- **Reutilização de código** – Código existente pode ser reutilizado. Se já existir uma classe semelhante àquela da qual precisamos, às vezes podemos dividir a classe existente e reutilizar algum código existente, em vez de implementar tudo novamente, só tomando cuidado com herança imprópria.

- **Manutenção mais fácil** – Uma alteração em um campo ou método compartilhado entre diferentes subclasses precisa ser feito somente uma vez.

- **Extensibilidade** – com a utilização da herança, torna-se muito mais fácil, de certa forma, estender uma aplicação existente.

14.4. DESVANTAGENS DA HERANÇA

- **Acoplamento** - Em Java não existe um limite para uso da herança, porém um sistema com muita herança deixa o código acoplado**.**

- **Quebra do encapsulamento** - E se precisamos acessar os atributos que herdamos? Não gostaríamos de deixar os atributos de Funcionario, public, pois dessa maneira qualquer um poderia alterar os atributos dos objetos deste tipo. Existe outro modificador de acesso, o **protected**, que fica entre o private e o public. Um atributo protected só pode ser acessado (visível) pela própria classe ou suas subclasses e até pelas classes do mesmo pacote. Mas não é uma boa ideia deixar que a classe filha acesse os atributos da classe mãe, muitos especialistas afirmam que a herança quebra o encapsulamento.

- **Herança múltipla** - Uma classe pode ter várias filhas, mas pode ter apenas uma mãe.

Viste o blog da Caelum, existe um artigo interessante abordando esse tópico: http://blog.caelum.com.br/2006/10/14/como-nao-aprender-orientacao-a-objetos-heranca/ James Gosling, um dos criadores do Java, é um crítico do mau uso da herança. Nesta entrevista ele discute a possibilidade de se utilizar apenas interfaces e composição, eliminando a necessidade da herança: http://www.artima.com/intv/gosling3P.html

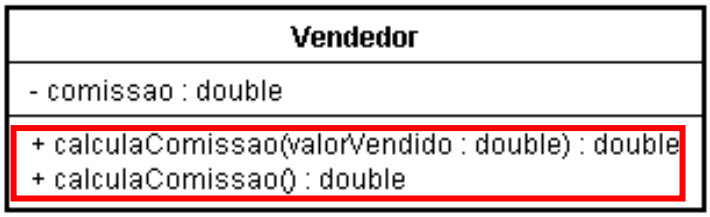
14.5. MÉTODOS NA HERANÇA

- ***Sobrecarregar*** um método é fornecer mais de um método com o mesmo nome, mas com diferentes assinaturas para distingui-los. Sobrecarregar um método herdado simplesmente significa que você adicionou um novo método, com o mesmo nome, mas com uma assinatura diferente, de um método herdado. O mesmo conceito aplicado em construtores.

- ***Sobrescrever*** um método significa substituir a implementação da superclasse daquele método com a sua própria. A sobrescrita ou reescrita de método está diretamente relacionada com herança e é a possibilidade de manter a mesma assinatura de um método herdado e reescrevê-lo na subclasse.

Exemplo de sobrecarga:

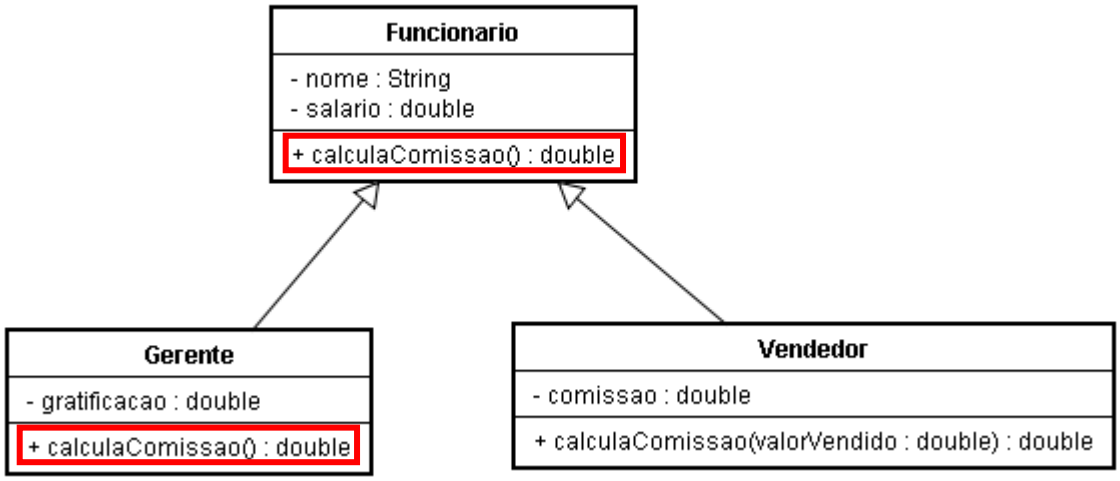
A sobrecarga de método é a possibilidade de implementar múltiplos métodos com o mesmo nome mas com uma lista de argumentos diferentes. A sobrecarga pode ser implementada tanto em métodos da mesma classe quanto em métodos herdados de uma superclasse. Sobrecarregar um método herdado simplesmente significa que adicionamos um novo método, com o mesmo nome, mas com uma assinatura diferente, de um método herdado. O mesmo conceito estudado em construtores.



No exemplo, a classe `Vendedor` possui dois métodos `calculaComissao`. Porém, o que ocorre é que o primeiro método calcula a comissão recebendo o valor vendido e outro método calcula a comissão e não possui parâmetros.

Exemplo de sobrescrita:

Como dito anteriormente, todo fim de ano, os funcionários recebem uma bonificação. Os funcionários comuns recebem 30% do valor do salário e os gerentes, 50%. Então o método calculaBonificacao da classe Gerente herdado da classe Funcionario não é adequado, a solução é a reescrita de método.



Observe que não é permitida a classe filha (Gerente) acessar a classe mãe (Funcionario) devido o uso do private nos atributos.

```
public class Gerente extends Funcionario{

    private double gratificacao;

    // getters e setters omitidos

    public double calculaBonificacao() {
        return this.salario * 0.5;
    }
}
```

Se alterá-los para protected seria permitido, mas quebro o encapsulamento. Para resolver use a palavra chave **super**.

```
public class Gerente extends Funcionario{

    private double gratificacao;

    // getters e setters omitidos

    public double calculaBonificacao() {
        return super.getSalario() * 0.5;
    }
}
```

Na chamada de um método sobrescrito o Java considera primeiro a classe a partir da qual o objeto foi instanciado, se a superclasse possuir um método com a mesma assinatura este será descartado.

14.6. A PALAVRA-CHAVE SUPER

Depois de reescrito, não podemos mais chamar o método antigo que fora herdado da superclasse, realmente alteramos o seu comportamento. Mas podemos invocá-lo no caso de estarmos dentro da classe.

Uma invocação de `super.método` sempre usa a implementação do método que a superclasse define (herda).

Imagine que para calcular a bonificação de um Gerente devemos fazer igual ao cálculo de um Funcionario porem adicionando R$ 1000. Poderíamos fazer assim:

O método calculaBonificacao da classe Funcionario continua calculando 30% do valor do salário para todos os funcionários.

```
public class Funcionario {

    private String nome;
    private double salario;

    // getters e setters omitidos

    public double calculaBonificacao() {
        return this.salario * 0.3;
    }
}
```

O método calculaBonificacao da classe Gerente é reescrito adicionando R$1000.

```
public class Gerente extends Funcionario{

    private double gratificacao;

    // getters e setters omitidos

    public double calculaBonificacao() {
        return super.getSalario()* 0.3 + 1000;
    }
}
```

Aqui teríamos um problema: o dia que o calculaBonificacao do Funcionario mudar, precisaremos mudar o método do Gerente para acompanhar a nova bonificação. Para evitar isso, o calculaBonificacao do Gerente pode chamar o do Funcionario utilizando-se da palavra chave super.

```
public class Gerente extends Funcionario{

    private double gratificacao;

    // getters e setters omitidos

    public double calculaBonificacao() {
        return super.calculaBonificacao() + 1000;
    }
}
```

Ao contrário das chamadas super dos construtores, o nome do método da superclasse é declarado explicitamente, pode ocorrer em qualquer lugar dentro desse método.

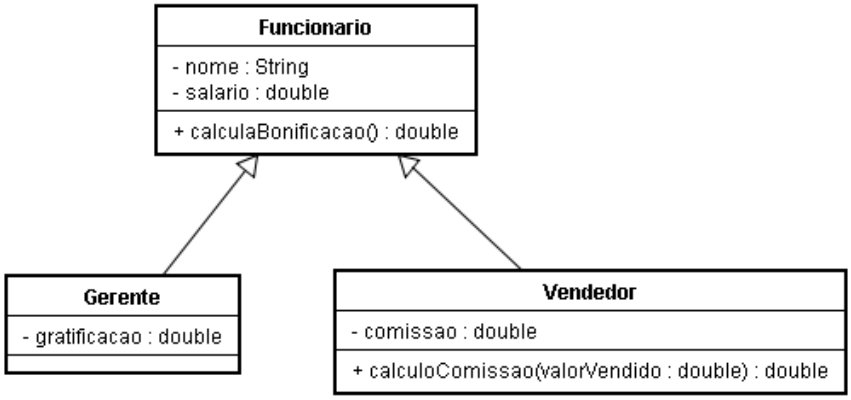
Essa é uma prática comum, pois muitos casos o método reescrito geralmente faz "algo a mais" que o método da classe mãe. Chamar ou não o método de cima é uma decisão sua e depende do seu problema. Algumas vezes não faz sentido invocar o método que reescrevemos.

14.7. EXERCÍCIO: Herança

1) Crie o projeto *ds1-jsf-heranca* que se encontra na pasta PROJETOS no moodle para sua área de trabalho.

2) Importe o projeto *ds1-jsf-heranca*:
   a. Acesse o menu **File ➔ Import**
   b. Selecione **General ➔ Existing Project into Workspace** e **Next**
   c. Clique em **Browse** e selecione a pasta do projeto
   d. Clique em **Ok** e, depois, em **Finish**.

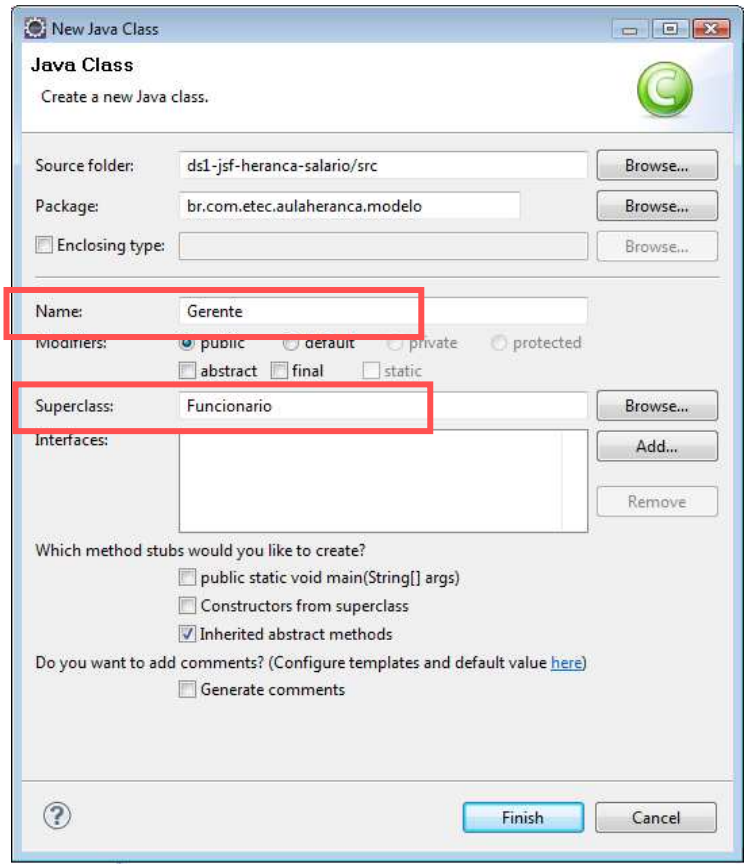
3) No pacote *br.com.etec.aulaheranca.modelo* crie as classes conforme diagrama de classes:



a) Crie a classe Funcionario, gere os getters e setters automaticamente. (CTRL + 3 e digite ggas)

```
1  package br.com.etec.aulaheranca.modelo;
2
3  public class Funcionario {
4
5      private String nome;
6      private double salario;
7
8      public String getNome() {
9          return nome;
10     }
11
12     public void setNome(String nome) {
13         this.nome = nome;
14     }
15
16     public double getSalario() {
17         return salario;
18     }
19
20     public void setSalario(double salario) {
21         this.salario = salario;
22     }
23
24     public double calculaBonificacao() {
25         return this.salario * 0.3;
26     }
27 }
```

b) Crie a classe Gerente, **New → Class →** no **Name** digite *Gerente* e no **Superclass** digite *Funcionario*.



c) Crie o atributo gratificação, get, set e o método calculaBonificacao que invoca o método da superclasse Funcionario e adiciona R$1000.

```
1  package br.com.etec.aulaheranca.modelo;
2
3  public class Gerente extends Funcionario {
4
5      private double gratificacao;
6
7      public double getGratificacao() {
8          return gratificacao;
9      }
10
11     public void setGratificacao(double gratificacao) {
12         this.gratificacao = gratificacao;
13     }
14
15     public double calculaBonificacao() {
16         return super.calculaBonificacao() + 1000;
17     }
18 }
```

d) Perceba que trata-se de um reescrita de método. Posicione o mouse no símbolo, é exibido o pacote e a classe que contém o método na superclasse.

```
14
15  overrides br.com.etec.aulaheranca.modelo.Funcionario.calculaBonificacao
16          return super.calculaBonificacao() + 1000;
17      }
18  }
```

e) Crie a classe Vendedor e estenda da classe Funcionario. O método calculaBonificacao invoca o método da superclasse Funcionario e adiciona R$500. Acrescente o calculaComissao que invoca o salário através da palavra chava super e invoca o método getSalario(), recebe o valor vendido do período por parâmetro, calcula e retorna o valor da comissão do vendedor.

```
1   package br.com.etec.aulaheranca.modelo;
2
3   public class Vendedor extends Funcionario {
4
5       public double calculaBonificacao() {
6           return super.calculaBonificacao() + 500;
7       }
8
9       public double calculaComissao(double valorVendido){
10          return  (valorVendido * 0.1);
11      }
12
13  }
```

4) Só para testar, crie uma classe *TestaHerança* no pacote *br.com.etec.aulaheranca.teste*:
   a) Instancie um objeto do tipo Gerente e altere o atributo salario e gratificacao através dos métodos setters.
   b) Instancie um objeto do tipo Vendedor e altere o atributo salario através do método set.
   c) Exiba a mensagem do cálculo do salário final no console.

```
public class TestaHeranca {

    public static void main(String[] args) {

        Gerente gerente = new Gerente();
        gerente.setSalario(2000);
        gerente.setGratificacao(200);
        double salarioFinalG = gerente.getSalario() + gerente.getGratificacao();
        System.out.print("Salário Final do gerente: " + salarioFinalG);

        Vendedor vendedor = new Vendedor();
        vendedor.setSalario(1000);
        vendedor.calculaComissao(1000);
        double salarioFinalV = vendedor.getSalario()
                + vendedor.calculaComissao(1000);
        System.out.println("\nSalário Final do vendedor: " + salarioFinalV);
    }
}
```

d) Execute a classe TestaHerança como JavaAplication. Verifique o resultado e conceitos aplicados da herança.

```
Console
<terminated> TestaHeranca [Java Application] C:\Pro
Salário Final do gerente: 2200.0
Salário Final do vendedor: 1100.0
```

14.8. POLIMORFISMO

O que guarda uma variável do tipo `Funcionario`? Uma referência para um `Funcionario`.

Na herança, vimos que todo Gerente é um `Funcionario`, pois é uma extensão deste. Podemos nos referir a um Gerente como sendo um `Funcionario`. Se alguém precisa falar com um `Funcionario` do banco, pode falar com um `Gerente`! Por que? Pois `Gerente` **é um** `Funcionario`. Essa é a semântica da herança.

```
Gerente gerente = new Gerente();
Funcionario funcionario = gerente;
funcionario.setSalario(5000.00);
```

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

Até aqui tudo bem, mas e se eu tentar:

```
funcionario.calculaBonificacao();
```

Qual é o retorno desse método? 2500 ou 1500? No Java, a invocação de método sempre vai ser **decidida em tempo de execução.** O Java vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não com a que estamos usando para referenciá-lo.

Apesar de estarmos nos referenciando a esse `Gerente` como sendo um `Funcionario`, o método executado é o do `Gerente`. O retorno é 2500.

Parece estranho criar um gerente e referenciá-lo como apenas um funcionário. Por que faríamos isso?

Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo `Funcionario`:

```
public class ControleBonificacao {
    private double totalDeBonificacoes = 0;

    public void registra(Funcionario funcionario) {
        this.totalDeBonificacoes += funcionario.calculaBonificacao();
    }

    public double getTotalDeBonificacoes() {
        return this.totalDeBonificacoes;
    }
}
```

E, em algum lugar da minha aplicação (ou no main se for apenas para testes):

```
ControleBonificacoes controle = new ControleBonificacoes();
Gerente funcionario1 = new Gerente();
funcionario1.setSalario(5000.0);
controle.registra(funcionario1);

Funcionario funcionario2 = new Funcionario();
funcionario2.setSalario(1000.0);
controle.registra(funcionario2);
System.out.println(controle.getTotalDeBonificacoes());
```

Repare que conseguimos passar um `Gerente` para um método que recebe um `Funcionario` como argumento.

Pense como numa porta na agência bancária com o seguinte aviso: "Permitida a entrada apenas de Funcionários". Um gerente pode passar nessa porta? Sim, pois `Gerente` **é um** `Funcionario`.

Qual será o valor resultante? Não importa que dentro do método registra do `ControleBonificacoes` receba `Funcionario`. Quando ele receber um objeto que realmente é um `Gerente`, o seu método reescrito será invocado.

Reafirmando: não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele.

E a classe Vendedor, por exemplo, que é filha de Funcionario, precisaremos mudar a classe de ControleBonificacoes? Não. Basta a classe Vendedor reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimorfismo, juntamente com a herança e reescrita de método: diminuir acoplamento entre as classes, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou ControleBonificacoes pode nunca ter imaginado a criação da classe Vendedor. Isso traz um reaproveitamento enorme de código.

14.9. EXERCÍCIO: Polimorfismo

1) No mesmo projeto ds1-jsf-heranca;

2) Crie a classse `ControleBonificacoes` no pacote *br.com.etec.aulaheranca.modelo*

a. Na classe *ControleBonificacoes,* crie um atributo private do tipo double com o nome totalDeBonficacoes e inicialize com 0.
b. Crie um método que registra todas as bonificações e recebe um Funcionario como parâmetro.
c. Crie o método get do atributo totalDeBonificacoes.

```
package br.com.etec.aulaheranca.modelo;

public class ControleBonificacoes {
    private double totalDeBonificacoes = 0;

    public void registra(Funcionario funcionario) {
        this.totalDeBonificacoes += funcionario.calculaBonificacao();
    }

    public double getTotalDeBonificacoes() {
        return this.totalDeBonificacoes;
    }
}
```

3) Só para testar, crie uma classe `TestaPolimorfismo` no pacote *br.com.etec.aulaheranca.teste*:

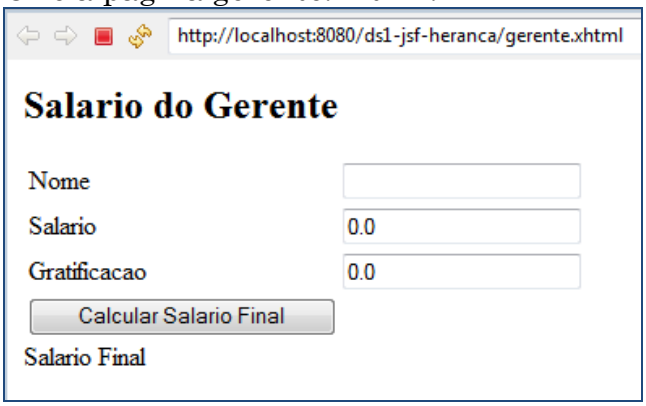
```
public class TestaPolimorfismo {
    public static void main(String[] args) {
        ControleBonificacoes controle = new ControleBonificacoes();
        Gerente funcionario1 = new Gerente();
        funcionario1.setSalario(2000.0);
        controle.registra(funcionario1);

        Vendedor funcionario2 = new Vendedor();
        funcionario2.setSalario(1000.0);
        controle.registra(funcionario2);
        System.out.println("Total da bonificacao: "
                + controle.getTotalDeBonificacoes());
    }
}
```

a) Execute a classe TestaPolimorfismo como JavaAplication. Verifique o resultado e conceitos aplicados da herança e polimorfismo.

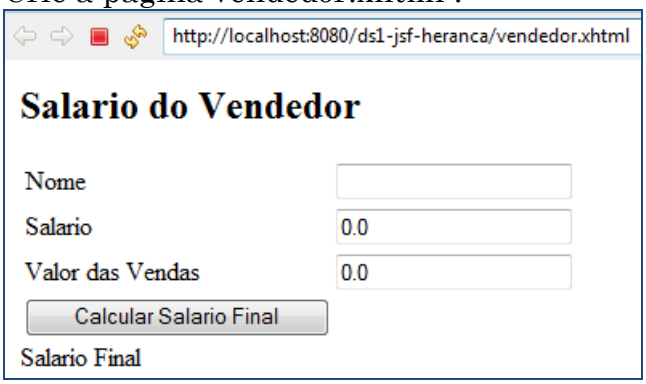
```
Console
<terminated> TestaPolimorfismo [Java Applica
Total da bonificacao: 2400.0
```

14.10. EXERCÍCIO: Projeto JSF: Herança e Polimorfismo

1) Crie a página gerente.xhtml :



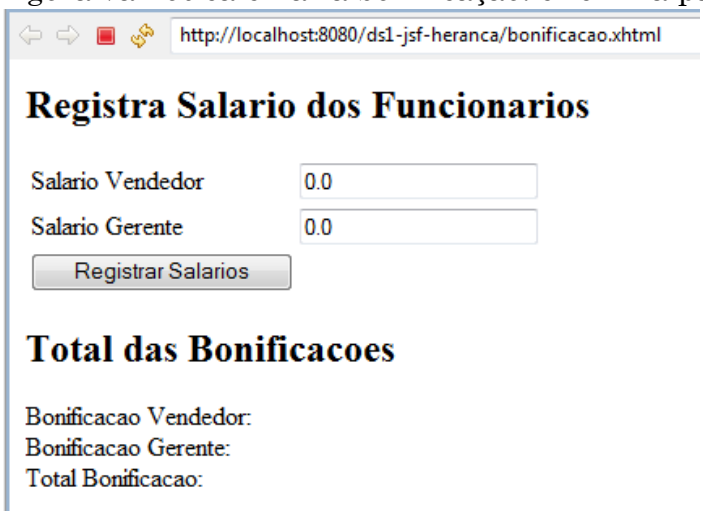
2) Crie a página vendedor.xhtml :



3) Crie o managed bean do Vendedor e Gerente com os atributos e métodos necessários.

4) Calcule o salário final do vendedor (salário + comissão) e do gerente (salário + gratificação) e exiba nas respectivas páginas.

5) Agora vamos calcular a bonificação: crie uma página `bonificacao.xhtml`:



6) Crie o managed bean da Bonificacao com os atributos e métodos necessários.

Referência Bibliográfica:

Apostila Caelum - http://www.caelum.com.br/
http://www.globalcode.com.br/home
http://www.k19.com.br/downloads/apostilas-java
http://www.algaworks.com/
Barnes, David J. e Kolling, Michael. Programação Orientada a Objetos com Java. Pearson
Arnold, Ken, Gosling, James e Holmes, David. A Linguagem de Programação Java. Bookman
Harvey M. Deitel, Paul J. Deitel. Java Como Programar. 6ed. Pearson
Flanagan, David. Guia Essencial do Java. Bookman
Martin, Fowler. UML Essencial.
Livro de Informática – Centro Paula Souza